PADL/SD-01

THE PADL-1.0/n PROCESSOR:
OVERVIEW & SYSTEM DOCUMENTATION

by

The Senior Staff

of the

PRODUCTION AUTOMATION PROJECT

October 1977
PADL SYSTEM DOCUMENT NO. 01

PRODUCTION AUTOMATION PROJECT
University of Rochester
Rochester, New York 14627

PADL/SD-01

THE PADL-1.0/n PROCESSOR:
OVERVIEW & SYSTEM DOCUMENTATION

by

The Senior Staff:

W. B. Fisher
E. E. Hartquist
A. A. Requicha
H. B. Voelcker

October 1977

The work reported in this document was supported primarily by the National Science Foundation under grants GI-34274X and APR76-01034.
SUMMARY

This is the first document in a series (the PADL/SD series) that provides system documentation for the PADL-1.0/n processor. The series is directed at system programmers, analysts, and software designers, not at ordinary users of PADL systems.

This document provides an overview of the PADL system and the PADL processor's internal organization. It also explains our approach to documentation and contains a guide to other relevant documents.
1. The PADL-1.0/n System ................................................. 1
   1.1 The input medium: PADL-1.0 ................................. 1
   1.2 Output media .................................................. 4
   1.3 The processor .................................................. 4
   1.4 Documentation ............................................... 7
   1.5 Dissemination ............................................... 7

2. The PADL-1.0/n Processor ............................................. 8
   2.1 Logical organization .......................................... 8
      2.1.1 An expanded view of the processor ............... 8
      2.1.2 Major transformations and structures ......... 10
      2.1.3 The control hierarchy ................................. 12
   2.2 Major information structures ............................... 15
   2.3 Major functional transformations ......................... 19
   2.4 Supporting systems .......................................... 23
      2.4.1 Implementation language: FLECS/FORTRAN ....... 23
      2.4.2 Subroutine packages .................................. 25
      2.4.3 Utilities ................................................ 26
   2.5 Computing system considerations ............................ 26
      2.5.1 The PADL processor's storage requirements .... 27
      2.5.2 Storage constraints in minicomputers ............ 28
      2.5.3 The RT-11 organization ................................. 29
      2.5.4 Alternative organizations for minicomputers .... 32
      2.5.5 The design of overlay structures .................. 32
   2.6 Commissioning the processor .................................. 36
      2.6.1 System dependencies .................................. 36
      2.6.2 A commissioning procedure ............................ 37
3. Documentation........................................39
   3.1 A hierarchical view of geometric modelling....39
   3.2 The approach to documentation......................40
   3.3 The PADL/SD series of reports...................42

References........................................45

Appendix: System dependent routines.................47
1. THE PADL-1.0/n SYSTEM

Fig. 1-1 depicts the PADL-1.0/n system. We shall discuss it in the traditional manner, i.e., in terms of its input and outputs, and the transformations effected by the Processor in Fig. 1-1 that relate them.

![Figure 1-1](image)

**INPUT MEDIUM**

PADL-1.0

**FIGURE 1-1**

The PADL-1.0/n System

1.1 THE INPUT MEDIUM: PADL-1.0

"PADL" is an acronym for Part and Assembly Description (or Definition) Language. PADL is precisely that—a stylized and extensible language for describing certain kinds of mechanical parts and assemblies in a very precise manner.

PADL admits two kinds of statements: 1) definitional statements for solid objects, and related declarations (e.g., material specifications), and 2) commands. Definitional statements are in assignment format, e.g.

\[
\text{\$PART = \$BLANK .DIF. \$B(A,B,C)} \\
\text{A = 4.00 : PM(.01)}
\]
whereas commands are in imperative format, e.g.

DRAW(8PART), 03R, DINTOL.

Broadly, the system merely "listens" to valid definitional and declarative statements, whereas commands elicit action. At present the command repertoire is oriented strongly toward the generation of drawings and CRT displays.

PADL statements may be entered or edited interactively through a keyboard, or read from files which typically contain whole definitions for parts. A processor for an interactive graphic version of the language, tentatively dubbed GPADL, is being developed, but it will not be covered in the current series of system documents. (Briefly, the GPADL system is a "front end" to a conventional PADL system that enables statements to be entered by graphic means rather than as text.)

The current version of the language, PADL-1.0, is relatively well documented. TM-20b [1] contains a formal specification, and the PADL Primer [2] serves as an informal user's guide.

* 

PADL is extensible in three different senses. One line of development is aimed at making PADL systems easier to use through the provision of "conveniences" -- graphic input media, for example. Another line of development seeks to extend PADL's range of practical applications, which are now predominantly graphic, into manufacturing and other areas. Extensions of this nature will affect mainly the language's command repertoire.

The third line of extension is concerned with PADL's intrinsic descriptive power, and affects the language's object definitional facilities. The nominal descriptive power of a constructive solid geometry (PADL-style) language depends on

1) the set of available primitive solids;
2) the nature of the operators available for combining solids;
3) the availability of rigid transformations (translation, rotation) for positioning solids prior to combination.

Some prospective versions of PADL are shown in Table 1-1. All are presumed to contain regularized set operators for
effecting intersection (.INT.), union (.UN.), and difference (.DIF.), and these operators are presumed to be general -- which means that they can be applied to any pair of solids. In addition, all versions are presumed to have dimensioning and tolerancing facilities equivalent to those specified in the current ANSI standards [3].

<table>
<thead>
<tr>
<th>Characterization</th>
<th>Designation</th>
<th>Primitive Solids</th>
</tr>
</thead>
<tbody>
<tr>
<td>Three standard</td>
<td>PADL-1.0</td>
<td>Blocks, Cylinders</td>
</tr>
<tr>
<td>orientations:</td>
<td>PADL-1.4</td>
<td>Blocks, Cylinders, Wedges &lt;1-1&gt;</td>
</tr>
<tr>
<td>$X$, $Y$, $Z$</td>
<td>PADL-1.6</td>
<td>Blocks, Cylinders, Wedges, Cones</td>
</tr>
<tr>
<td>(&quot;orthogonal</td>
<td>PADL-1.8</td>
<td>Blocks, Cylinders, Wedges, Cones, Spheres, Tori</td>
</tr>
<tr>
<td>positioning&quot;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unrestricted</td>
<td>PADL-2.0</td>
<td>Blocks, Cylinders &lt;1-2&gt;</td>
</tr>
<tr>
<td>orientation:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(&quot;general</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rotation&quot;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PADL-2.8</td>
<td>Blocks, Cylinders, Cones, Spheres, Tori</td>
</tr>
</tbody>
</table>

TABLE 1-1

Prospective Versions of PADL

* * *

<1-1> Four of each Wedge's five planar faces are perpendicular to a principal axis.

<1-2> A Wedge primitive may be defined constructively by intersecting two Blocks.
1.2 OUTPUT MEDIA

The current PADL-1.0/a system can produce line drawings of defined objects, in a variety of styles, on either of two devices: an online CRT, and an offline plotter. The following principles were adopted to maximize transportability of the system by minimizing its dependence on specific peripheral devices.

1) Both devices are viewed as passive (write-only).

2) CRT displays are generated by a small set of well defined, low level graphic utilities that are device dependent. A variety of CRT terminals can be interfaced to the system by providing appropriate sets of utilities for each.

3) The production of drawings on plotters is done through an intermediate file created by the PADL processor. This file is "post-processed" by a separate program to generate commands to drive a specific plotter. The file structure, the post-processor structure, and plotter-specific routines are defined in the PADL system documentation.

PADL source definitions stored as named files constitute a third output available from the system. The current processor relies mainly on file-handling facilities in its host computer's operating system to administer both definition and drawing files.

Various printed and displayed alphanumeric outputs, e.g. listings of PADL source definitions and characteristic parameters of object boundaries, are also available: consult the PADL Primer [2].

* * *

1.3 THE PROCESSOR

From a user's point of view, the PADL processor performs two functions: 1) it detects lexical and syntactic errors in input statements, and 2) it executes commands to produce output. Section 2 of this document provides a high level description of the processor's organization and operation, and thus we shall merely summarize a few salient features.

The processor maintains, as shown in Fig. 1-2, two internal representations of a defined object in addition to the PADL source strings: 1) a tree-structured (constructive) representation which is, in essence, merely a more compact and convenient version of the source definition, and 2) a
&B1 = $B(...)

... &P = &P1.DIF, &C1

DISP(&P)

FIGURE 1-2
PADL Processor
representation of the object's boundary as a collection of bounding "faces". The first is the primary definition. The second is derived, by a process called boundary evaluation, when it is needed, e.g. to provide data for generating graphic displays. The current processor has four major subsystems: a language translator, a boundary evaluator, a graphic output generator, and a "DTA Processor" to check and display dimension, tolerance, and mechanical-attribute information.

The PADL processor is based on mathematical principles and algorithms new to the engineering world, and thus it is not surprising that improvements in processor technology are being made almost continuously. The "/n" suffix in a processor's designation signifies its relative development level in the following manner.

```
PADL-1.0/n1.n2

boundary

evaluator

designation

availability & sophistication of other facilities
```

1) Larger n1, n2 values signify "better".

2) In the current generation of graphically oriented processors, the efficiency of a processor's boundary evaluator is the major determinant of the processor's efficiency. Thus a processor designated /2.n2 is faster than a /1.n2 processor. <1-3>

3) The "other facilities" designation is used to indicate the relative completeness of the implementation.

All processors developed to date have been written in FLECS, a Structured FORTRAN dialect developed at the University of Oregon [5]. FLECS programs are translated by a preprocessor into standard FORTRAN for conventional compilation.

PADL processors are relatively large and complex. Typically they contain 250-300 subroutines, 20,000-plus lines of source code, and require approximately 250-300 Kbytes of

-----------

<1-3> All PADL boundary evaluators developed to date have been members of the ENDRUN family. The name, which is P.A.P. patois, denotes a generic family of algorithms based on set membership classification principles [4].
storage if executed in a non-overlaid environment. However, considerable effort has been devoted to making the prototype processor modular and the inter-process communication patterns partitionable. As a result, it is possible to design overlay structures that enable the processors to execute (at reduced speed) in 16-bit minicomputers having as little as 64 Kbytes of storage. See Section 2.5 for more information.

* *

1.4 DOCUMENTATION

Section 3 discusses the general problem of documenting PADL technology and its underlying mathematical principles and algorithms. PADL/SD reports are but one component of our solution to that problem.

Documents in the PADL/SD series provide system documentation for the PADL-1.0/n processor. The major purpose of the series is to explain how the various entities, operators, and commands which appear in the PADL language are defined mathematically and embodied (implemented, represented) in data structures and procedures within the processor.

The SD series is directed at system programmers, i.e. those who must work within the PADL processor to modify or repair it, and at designers and implementors of more advanced systems. Ordinary users of PADL are not encouraged to read SD's. Readers are assumed to be proficient in the use of PADL, FORTRAN, and FLECS [5]. Some SD's require, in addition, some specialized mathematical knowledge and familiarity with software support systems (CPAK, TPAK, STGPAK) described in Section 2.

* *

1.5 DISSEMINATION

The conditions governing dissemination of the PADL-1.0/n processor and its associated documentation are set forth in Reference 14.

* * *
2. THE PADL-1.0/n PROCESSOR

2.1 LOGICAL ORGANIZATION

The organization of the PADL processor will be explained using a number of "system" diagrams. The P.A.P.'s conventions for such diagrams are shown in Fig. 2-1: rectangles denote artifacts ("things"), and circles denote transformations.

More precisely, rectangles denote "information structures" which may be mathematical representations (e.g. equations defining particular geometric entities) or computer representations -- arrays, lists, and similar data structures. Transformations are, correspondingly, mathematical mappings (often defined as mathematical algorithms) or computer procedures. Section 3.1 contains a discussion of the role of these types of representations and transformations in hierarchical modelling.

![Diagram](image)

**FIGURE 2-1**

Diagrammatic Conventions

2.1.1 AN EXPANDED VIEW OF THE PROCESSOR

Figure 2-2 is a more detailed version of Figure 1-2. There are four major processes that transform information
FIGURE 2-2
An Expanded View of the Processor
structures to other information structures. (2-1) Brief descriptions of the various processors and structures are given in Sections 2.2 and 2.3.

What is not shown particularly well is the sequence in which processors are activated. Imagine that a user commands that a defined PDL object (i.e., an object for which trees and tables exist) be drawn for the first time. Presumably processors are evoked in the order: 1) boundary evaluator, 2) DTA evaluator, 3) graphic output generator. Now suppose that the user commands a second view of the same object. Will the processors be evoked in the same order? They could be, but the boundary file and the DTA trees and tables already exist. Hence only the graphic output generator need be evoked.

The moral? There are at least two useful views of the processor. In the sequel we shall outline first what the processor "accomplishes" in terms of major functional transformations on information structures, and then we shall describe the processor in terms of its control hierarchy.

* *

2.1.2 MAJOR TRANSFORMATIONS & STRUCTURES

The right side of Figure 2-3 shows the processor's major functional transformations; its major information structures are on the left. Each transformation should be viewed as a well defined mapping that takes well defined representations, i.e., symbol sequences, into (other) well defined representations. The overall result is the mapping of STRINGS to GRAPHIC OUTPUT.

Notice that the input processor of Figure 2-2 has become two processors, a definition translator and a command translator. Each has unique domains and ranges. The graphic output generator has also been partitioned into separate processors to handle tasks like "layout" (of drawings and CRT displays), "draw", and "post" (display DTA information graphically). Finally a new processor, the definition writer MAKDEF, has been added.

----------

(2-1) We shall often use the words "transformation", "process", and "processor" interchangeably, since (mathematical) transformations are implemented as processes executed by processors.
FIGURE 2-3

Major Functional Transformations
A partially ordered set of transformations that is sufficient for each command can be deduced by working backwards in Fig. 2-3, under the constraint that a processor's inputs must be available before it may be evoked.

Example: The following partial ordering is sufficient to produce an undimensioned drawing of a PADL object.

\[ \text{GDRAW} \triangleright \text{LAYOUT} \triangleright ((\text{BEVAL} \triangleright \text{DEFN-\text{XLATOR}}), \text{CMD-\text{XLATOR}}) \]

The ordering was derived left to right, by applying the rule "post-conditions \( \Rightarrow \) preconditions", to the relations specified by Fig. 2-3. It can be viewed as an internal command sequence that is to be executed right to left.

As remarked earlier, sufficient sequences often are not necessary sequences.

The various structures and transformations of Figure 2-3 will be described individually in sections 2.2 and 2.3. More detailed descriptions are given in other SD's.

* * *

2.1.3 THE CONTROL HIERARCHY

Figure 2-4 shows the hierarchy through which the application of transformations is controlled. At the top of the hierarchy is the system controller. When the system has been initialized by its Custodian, it alternates between "listening to the user" via the input subsystem and producing output via other subsystems. These other subsystems, each headed by a controller, correspond to a PADL system's major command capabilities. Since the only "significant" commands in the current version of PADL are DRAW and DISPLAY, there is but one application controller -- for graphics.

\[ \text{\ldots...} \]

\[ <2-2> \] The symbol "\( \triangleright \)" denotes that the right member precedes the left member. The symbol "\( \not\triangleright \)" denotes lack of precedence.

The example has been simplified somewhat by ignoring, for reasons implied in Section 2.3, the presence in Fig. 2-3 of the process MAKDEF.
Each of the major functional transformations shown in Fig. 2-3 appears as a process below the controller level, and each can be considered a controller of subprocesses. Near the bottom of the control hierarchy are supporting systems such as STOPAK for manipulating strings, and CPAK for handling geometric data. At the very bottom of the hierarchy are an extended memory system for handling array data (it is named HICAAR and is often referred to as "high core"), the FORTRAN library, and servicing routines for peripheral devices ("device handlers").

In an optimized system, controllers at the various levels are optimal planners and schedulers of subordinate processes. They "know" the partial order relationships implied in Fig. 2-3 (and similar relationships at lower levels), and they have means for tracking the status (i.e. currency or obsolescence) of the various information structures. They use this information to plan (construct, specify) sequences of transformations for each command, and to schedule processor evocations to execute the plan in a manner that optimizes the overall system's performance. (Planning and scheduling usually are not independent activities in optimized systems.)

The PADL-1.0/n processor is not optimized. Its controllers were developed in an evolutionary manner and are based on a series of preplanned sequences that exploit only a portion of the flexibility inherent in the hierarchical organization. We view the design and implementation of more flexible controllers as an interesting and solvable problem.

*

The foregoing brief discussion of control suggests one good reason (efficiency) for defining a system's functional transformations and control hierarchy as fully and as clearly as possible. There is another strong argument for careful definition that played a central role in the design of the current processor.

Succinctly, the PADL-1.0/n processor is not a small program; it must be overlaid if it is to execute in a small computer. By organizing the PADL processor into "layers" of processes that correspond to functional transformations, one hopes to arrive at a configuration in which

1) each major task is accomplished without having to recall major subsystems repeatedly,

2) modules on each level are appropriately short, and
3) no module need leave behind, for use by another module, lengthy but only locally relevant data.

We shall have more to say about this topic in a later section.

* * *

2.2 MAJOR INFORMATION STRUCTURES

This section describes briefly the major information structures shown in Fig. 2-3. The simplified descriptions which follow generally do not depict faithfully particular data structures in the PADL-1.0/n processor. The processor's actual data structures are described in the SD's cited as References.

STRINGS

'String' is the generic name used to denote a finite sequence of literal symbols -- ASCII characters, in our case. PADL definitional statements and commands are strings. The information structure STRINGS in Fig. 2-3 is actually a subsystem for processing strings, as shown in Fig. 2-5.

FIGURE 2-5

An Expanded View of the "STRINGS" Information Structure

The 'Input' in Fig. 2-5 refers to PADL statements supplied by users. These may be entered through any device that can accept character strings, e.g. a card reader or
keyboard. The 'String Processor' (not a distinct module in
the actual processor) separates command strings from
definitional strings and makes each type available to its
functional processor. It also saves definitional strings so
that object definitions may be listed, edited, and stored on
tape or disc.

OBJECT TREES AND TABLES
Reference: SD-06

These are 'high level' representations of constructively
defined objects. The primary structure is a binary tree --
see Fig. 2-6 -- whose nodes are operators (.UN., .DIF., etc.)
and whose subtrees are primitive solids or compositions.
Binary trees are represented in the processor via postfix
strings supplemented by tables of primitive solids, d-chains,
and distances.

One such representation is called the archival
definition; it faithfully represents user's input statements
and is considered the primary definition. The archival
definition is relatively verbose. More concise postfix
representations are constructed from the archival definition
for use by functional processors, e.g. BEVAL (the Boundary
Evaluator).

COMMAND VECTORS
Reference: SD-07

A command vector is the parsed form of a command string
supplied by a user. The major application controllers consult
Command Vectors to determine the nature of the high level
tasks that are to be accomplished.

BOUNDARY FILES
Reference: SD-10

An object's boundary file (b-file) represents the object
in terms of its bounding surfaces, as shown in Figure 2-7.
B-files are organized hierarchically in terms of object-level
data (e.g. the object's minimal enclosing box), surface-level
data, and edge-level data. Each bounding "face" is described
by face related data and edges. Each edge is characterized by
defining data and the two faces that "share" it.

The term b-file refers to the boundary file of a
particular object. In PADL processors dubbed /2.n and higher,
b-files are managed by a system called BFILE/m.
**FIGURE 2-6**

Object Trees and Tables
"DTA trees and tables" refers to a collection of relations between user supplied technical data and an object's geometric features. The most prominent class of relations, dimension trees (d-trees), will serve as an example.

Sets of referential ("chained") dimensions may be represented by tree structures if the rules of conventional dimensioning practice are observed. Fig. 2-8 illustrates the representation. Observe that each node of the d-tree corresponds to a face (or faces), while the tree's branches correspond to the directed distances between these faces. The d-tree relations are represented internally as list structures whose atoms are "names" for b-file faces and entities in distance tables.

VIEW PARAMETERS

This small information structure contains parameters which control the allocation of space in graphic displays, and GPAK <xfm>'s which define relationships between the PADI.
FIGURE 2-8
Conventional Dimensioning and d-trees

cordinate system and graphic display spaces for particular views.

GRAPHIC OUTPUT

The "Graphic Output" rectangle in Fig. 2-3 actually represents a pair of graphic subsystems, as shown in Fig. 2-9. Selection of the appropriate subsystem is determined by a Command Vector datum. The CRT Display File is never seen by PADL users, whereas the Drawing File often may be written to tape for remote postprocessing and plotting.

* *

2.3 MAJOR FUNCTIONAL TRANSFORMATIONS

This section describes briefly the major functional transformations shown in Fig. 2-3. The descriptions are highly simplified and in some cases deviate from the implementation for the sake of brevity and clarity. See the
SD's cited as References for more detailed information.

DEFINITION TRANSLATOR

The syntactic rules underlying PADL's definitional facilities form a strict operator precedence grammar. <2-3> The module labelled Definition Translator in Figure 2-3 contains a table driven parser that re-represents definitional input statements in terms of <primitive-solid>'s, <d-chain>'s, <dist>'s, <tol-spec>'s, <number>'s, and so forth. The tree- and table-structured archival definitions discussed earlier are constructed from these entities by a data management module evoked by the Translator.

COMMAND TRANSLATOR

PADL commands fall into two categories: those that cause

<2-3> This is true for the 1977 version of the language defined in TM-20b [1], but not for the 1974 version of the language defined in TM-20a [6]. The 1977 version is a syntactically regularized superset of the 1974 version with more explicit data types. From an unsophisticated user's point of view, the differences between the versions are almost undetectable.
major application subsystems to be evoked, and "housekeeping" commands -- SHOW, LIST, and so forth. The Command Translator processes both. Data parsed from major commands (of which there are only two at present, DRAW and DISPLAY) are deposited in the Command Vectors structure for use by the appropriate command executors, whereas the execution of housekeeping commands is controlled within the input subsystem.

DEFINITION WRITER (MAKDEF) Reference: SD-05

Certain commands require that the PADL processor be able to construct internal definitions of "new" objects from definitions of "old" objects supplied by users. The module MAKDEF does this in two different contexts.

One context is sectioning, i.e. the generation of data sufficient to display an object that has been sectioned ("cut open") by a user-specified plane. MAKDEF generates an explicit constructive definition of the sectioned object for processing by the standard display procedures. The second context is the display of nominal "volumes of interference" in objects defined as assemblies. Again, MAKDEF generates an explicit constructive definition of the interference volume by generating all distinct pairwise .INT.'s of the assembly's component parts.

MAKDEF is a small module with modest capabilities at present, but it is almost certain to grow in size and importance as PADL's capabilities expand.

BOUNDARY EVALUATOR (BEVAL) Reference: SD-09

BEVAL accepts a terse constructive definition of an object in postfix notation and produces a boundary file (b-file; see Fig. 2-7). On a conceptual level, there are two major subprocesses at work. One generates a sufficient set of "candidate entities" for inclusion in the b-file, and the other "classifies" each candidate entity and inserts in the b-file only those portions that are in the boundary of the object. (2-4)

(2-4) All boundary evaluators in the ENDRUN family (see Footnote (1-3)) operate on these principles, but the organization and coupling of the generation and classification processes change markedly when one moves from the /1.n and /2.n processors to the later processors. See SD-09 and Reference 4 for further information.
Entities called "pfaces", which are bounded faces of the primitive solids in an object's definition, are the primary candidates. (It can be shown that each "face" in an object's boundary can be associated with one or more pfaces, and hence the set of pfaces is a sufficient set of candidate entities.) Each pface is classified in a somewhat indirect manner. Specifically, b-file faces are treated as sub- or supersets of pfaces defined by bounding edges. Thus for each pface a sufficient set of candidate edges is generated, and each candidate edge is classified (i.e. tested for membership in the object's boundary). Only pface sub- or supersets defined by proper sets of bounding edges are entered in the b-file as faces.

DTA EVALUATORS (DTEVAL) Reference: SD-11

DTA Evaluators construct, and test the consistency of, relations between b-file entities (geometric "features" of objects, e.g. particular faces) that are specified via PADL's facilities for expressing tolerances, relative dimensions, and attributes.

At present there is but one such evaluator, DTEVAL. It builds a d-tree for each primary direction in the (SX, SY, SZ) coordinate system from relational data supplied by the user in the form of d-chains. A d-chain may be viewed as specifying a portion of a d-tree by describing a path from one feature (tree node) to another. DTEVAL merges these portions of d-trees into single trees having no redundant surfaces at its nodes and no redundant branches that connect nodes. DTEVAL also checks each tree to verify that each valid surface in a b-file is linked dimensionally to another valid surface and not to a non-existent ("phantom") surface.

LAYOUT Reference: SD-13

LAYOUT is a graphic support process that plans the arrangement of material in displays or drawings. LAYOUT "knows" the more important conventions of traditional engineering graphics, and it can determine (from Command Vector data, the b-file, etc.) the views a drawing is to have, whether they are to be dimensioned, what scaling factors are needed, and so forth. It uses this information to generate display area specifications for the graphic processes that actually produce displays and drawings.

GDRAW Reference: SD-13

GDRAW produces single views of objects by constructing a set of graphic lines and curves to represent an object's
surface (which is defined by its b-file) as seen from a specified viewing point. Subordinate routines — GLINE, GARC, GSPLIN — are used to display the graphic entities via calls to even lower level device dependent routines.

Many of the graphic lines and curves produced by GDRAW correspond directly to edges in the object's b-file, but others — notably "profile lines" for curved surfaces — have no direct b-file counterparts and must be generated by GDRAW. GDRAW also does certain kinds of hidden surface and hidden line elimination, and provides various graphic embellishments, e.g. hatching on sectioned surfaces.

DPOST

DPOST places ("posts") DTA information on displays and drawings after the information has been derived and/or checked by DTEVAL. DTA posting is surprisingly complex if one seeks (as in the current PADL system) to conform to the spirit of industrial drafting practices, since (for example) center lines, leader lines, and extension lines must be provided as well as dimension lines and related text. The rules determining the placement of lines and text are numerous and partially ad hoc, but the results are quite good in uncrowded drawings.

* *

2.4 SUPPORTING SYSTEMS

The PADL-1.0/n processor is a large computer program — actually a small main or master program and a large collection of subprograms — written ("implemented") in a standard programming language. In this section we shall discuss briefly the implementation language and its processors, and the two categories of subprograms that can be considered "general supporting systems" because they were not developed specifically for the PADL processor.

2.4.1 THE IMPLEMENTATION LANGUAGE: FLECS/FORTRAN

Portability (2-5) was one of the primary criteria governing the design of the PADL processor. Only two

---------------

(2-5) In computer circles "portability" means, ideally, that a program can be executed without modification on a variety of computer and operating system combinations.
mechanisms for attaining portability are known. The most popular by far, and the one we adopted, requires that a standardized and popular programming language be used as the implementation language. The other mechanism is based on meta languages and emulation and will not be discussed further.

There are only two programming languages, FORTRAN and COBOL, that are standardized and popular, i.e. widely used for production (as contrasted to research) purposes and supported by most computer vendors. We selected FORTRAN because it is the language most widely known and used in engineering circles, and because it is better suited to our applications than COBOL. Unfortunately, FORTRAN leaves much to be desired as a programming medium for sophisticated systems because its "expressive power" is quite limited. To elaborate briefly --

The "expressive power" of a programming language can be characterized in terms of the data types and the control structures that the language offers. A language's data types (e.g. real numbers, character strings) determine the entities that one can "talk about" in a direct manner, whereas a language's control structures (e.g. IF-THEN, INVOKE-PROCEDURE constructs) determine "what one can say" about data. Effective programming depends strongly on effecting a match between a problem's abstract structure and the expressive power of a programming language.

We elected to use two kinds of "FORTRAN extenders" to enhance FORTRAN's deficient (for our purposes) range of data types and control structures. One group of extenders -- suites of subroutines that serve to increase FORTRAN's range of data types -- is discussed in Section 2.4.2. The other extender, the Oregon FLECS system [5], is aimed at the control structure problem.

FLECS (Fortran Language with Extended Control Structures) is one of several FORTRAN dialects (supersets) that have been devised in the past few years to facilitate structured programming [7] in FORTRAN environments. FLECS programs are translated into FORTRAN for conventional compilation by a preprocessor written mainly in FLECS/FORTRAN. FLECS translators for use on a variety of computer and operating system combinations can be obtained from:

Computing Center
University of Oregon
Eugene, Oregon 97403
Attn: Terry Beyer
The entire PADL processor, with the exception of a few system dependent utilities cited in Section 2.6, is written in what might be termed "ANSI Standard FLECS". (That is, the translations of PADL modules are in ANSI Standard FORTRAN [8].)

* *

2.4.2 SUBROUTINE PACKAGES

The PADL processor relies on several self-contained "packages" of subroutines to provide data types not available in FORTRAN. Each package described below contains routines for declaring, storing, fetching, transforming, and "killing" its particular kinds of entities. All were designed and implemented by the Production Automation Project, and all share a common approach to data type enhancement that is based on the use of reserved INTEGER "names" for new-type entities. All are, or will become, publicly available.

STGPAK

STGPAK handles ASCII character strings. It contains procedures for reading and writing strings, locating particular strings immersed in text, finding the lengths of strings, copying and concatenating strings, and so forth. STGPAK is used most heavily by PADL's input subsystem.

GPAK

GPAK provides some modest but useful facilities for 3-D geometry and graphics. Its data types are vectors (REAL 3-tuples), duos (pairs of vectors), and transforms (REAL 16-tuples treated as 4 x 4 matrices). The geometry routines provide facilities for scaling, translating, rotating, and perspectively projecting vectors (points) in Euclidean 3-space, and for computing such vector attributes as cross and dot products.

The PADL-1.0/n processor does not use GPAK's graphic facilities, but GPAK's vectors and duos are used for a variety of representational purposes, and its transformational facilities are used in PADL's graphic output subsystem.

TPAK

TPAK deals with trees whose nodes may have a variable number of sons, and whose leaves are integer-valued atoms ("names", pointers, indices, etc.). TPAK offers procedures for constructing, traversing, editing, and testing trees. It
is used extensively for processing d-trees in the DTEVAL module, and it may be used in the future for generalizing the representations in the Object Trees & Tables structure (see Fig. 2-3) and elsewhere.

BFILE

BFILE (see section 2.2) is listed here because it can be thought of as a self contained supporting system. Its general utility is questionable, however, because it is strongly oriented toward the b-file representation scheme used in the PADL-1.0/n processor.

* *

2.4.3 UTILITIES

"Utilities" denotes the general purpose and generally low level facilities shown on the bottom of the hierarchy in Fig. 2-4. All are usually dependent on the particular computer, operating system, and peripheral devices that are used to support the PADL processor.

The FORTRAN LIBRARY (of I/O and mathematical routines) is self explanatory. "Device Handlers" include servicing routines for such common sequential and file-organized peripherals as a keyboard, printer, and a disc, and also routines to service any online graphic devices, e.g. a CRT which may or may not support graphic input.

HICAAR is a simple extended memory system for handling array-type data. (HICAAR is a mnemonic for High Core Allocation and Accessing Routines.) It is used primarily to support the data-type enhancement packages discussed above, but it is by no means limited to support of these packages. HICAAR is replaceable by any equivalent means for managing large arrays.

* *

2.5 COMPUTING SYSTEM CONSIDERATIONS

This section discusses the organization and characteristics of the PADL processor in terms of its requirements for computing system "capacity". The major concern is storage for data and executable code.

We shall orient much of the discussion toward 16-bit minicomputer installations because they expose starkly the key problem of program and data segmentation. The problem is
critical in minicomputer environments, but important also in large-computer virtual memory environments.

The availability of basic system software -- a FORTRAN compiler, execution monitor, linker, etc. -- is presumed. Hardware options that affect only the PADL processor's speed, e.g. floating point registers, will not be discussed.

* *

2.5.1 THE PADL PROCESSOR'S STORAGE REQUIREMENTS

Table 2-1 below summarizes the approximate storage requirements of the prototype PADL-1.0/n processor. <2-6>

<table>
<thead>
<tr>
<th>Executable Code</th>
<th>160 Kbytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>120 Kbytes</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>280 Kbytes</strong></td>
</tr>
</tbody>
</table>

**TABLE 2-1**

PADL-1.0/n Object Code and Data Areas

The table does NOT include allowances for a FORTRAN library, an execution monitor, device handlers, and a working stack or equivalent scratch space. Moreover, the Executable Code value is compiler dependent and the Data allowance reflects an upper bound on the complexity of objects which the system can accommodate.

More specifically, the 160 Kbyte Code value pertains to the Digital Equipment Corporation RT-11/FORTRAN compiler (Ver. 2C) for a PDP-11/40 computer equipped with an extended instruction set and a micro-coded floating point set. The cited compiler produces "threaded code" <2-7> which is

----------

<2-6> 1 byte = 8 bits, and for present purposes, 1K = 2**10 = 1024.

<2-7> "Threaded code" contains many calls to system subroutines which execute interpretively low level functions such as array accessing and arithmetic stack manipulation.
somewhat more conservative of storage but slower than register oriented code.

The Data allowance includes a relatively fixed allocation for general system use and an adjustable allocation for the storage of object trees, tables, and b-files. The 120 Kbyte allocation can accommodate objects whose total primitive solid count does not exceed 50, and whose "surface complexity" is commensurate with the primitive-count limit. This allocation should be more than sufficient to cover the majority of industrial parts that are describable in PADL-1.0. [12]

* *

Given the approximate storage requirements outlined above, the essential problem is: how should the source code be organized such that the object code can be executed efficiently in a minicomputer environment? We shall summarize some characteristics of minicomputers before we discuss a line of solution to the problem.

* *

2.5.2 STORAGE CONSTRAINTS IN MINICOMPUTERS

Single 16-bit computer words can be used to address randomly any one of 64K storage locations. In word-organized memories, 64K words can be addressed directly; in byte-organized memories, 64K bytes or 32K words can be addressed directly. This fact, plus the fact that most currently popular minicomputers do not include multiple word addresses in their instruction repertoires, influences the organization of minicomputer software in both obvious and subtle ways.

Many 16-bit computers partially circumvent their addressing limitations by providing special hardware to service any one of several different blocks of storage, each of which may contain up to 64K locations. Such computers may be viewed as several different 64K computers which share a processor, as a single 64K computer having additional blocks of fast-access "extended storage", or as various combinations of these two views.

Which view one takes is conditioned strongly by the sophistication of the operating system at one's disposal. A few operating systems for minicomputers offer true virtual memory facilities, wherein a programmer need not concern himself with size limitations at all -- until he discovers that his program is running slowly due to "thrashing"
(repeated exchanges between a disc file and the computer's main store). Other operating systems offer multi-tasking, which is convenient when several small programs are to execute "concurrently". The simplest (disc oriented) minicomputer operating systems offer only single-program or foreground/background monitors, plus automatic overlaying facilities. Special-purpose software must be written to use more than the basic 64K of fast random access storage. We shall call such additional storage "extended storage (memory)".

We shall discuss the PADL processor's organization in the environment associated with the last mentioned simple operating systems because

1) it is the environment we are most familiar with,

2) properly designed programs execute relatively efficiently (given the storage constraints) in such environments, and

3) it is probably the most difficult environment in which one can seek realistically to mount a PADL processor.

The prototype processor was designed to execute on a PDP-11/40 under RT-11, and we shall use its organization to illustrate the issues.

* * *

2.5.3 THE RT-11 ORGANIZATION

Table 2-2 is an expanded version of Table 2-1 which distinguishes between resident code and overlaid code, and between resident data and data in extended storage. A resident library is also included.
<table>
<thead>
<tr>
<th>Resident Executable Code</th>
<th>.5 Kbytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resident FORTRAN Library</td>
<td>11.5 Kbytes</td>
</tr>
<tr>
<td>Overlaid Executable Code</td>
<td>160.0 Kbytes</td>
</tr>
<tr>
<td>Resident (COMMON) Data</td>
<td>6.0 Kbytes</td>
</tr>
<tr>
<td>Extended-Memory Data</td>
<td>114.0 Kbytes</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>292.0 Kbytes</td>
</tr>
</tbody>
</table>

**TABLE 2-2**

Resident and Overlaid Object Code and Data

Fig. 2-10 shows how the allocations in Table 2-2 are mapped onto an extended-store PDP-11/40 operating under RT-11. A few facts about the host system are needed to interpret Fig. 2-10.

Firstly, although the 11/40 can contain up to 248 Kbytes of fast (core) storage, programs may execute only in the first 56 Kbytes. After provisions are made for the Monitor, Overlay Map, etc., only 25 Kbytes remain in which PADL's processes can execute. They do so, of course, by overlaying one another.

Programs cannot execute in the storage area labelled "High Core" in Fig. 2-10, but this area can be used very effectively as a fast random access store for data. Because programs executing in "low core" cannot address High Core directly, a small set of subroutines -- the HICAAR system cited in Section 2.4.3 -- is provided for storing data into and fetching data from High Core. (HICAAR contains FORTRAN callable assembly language subroutines which exploit the 11/40's memory management hardware.)

As Fig. 2-10 shows, most of High Core is allocated to data managed by the packages discussed in Section 2.4.2. Allocations can be changed during initialization of HICAAR, and the capacity of High Core can be expanded to about 200 Kbytes to accommodate more data. Alternatively, "old" data can be stored on disc when High Core is in danger of saturating ... and High Core then plays the role of a large buffer store for data.

* It should be evident that the RT-11 organization just described can be modified easily to run on large computers in about 250-300 Kbytes of storage. One need only concatenate the executable code into a single large block and replace the present HICAAR routines with subroutines which access FORTRAN
**SD-01**

---

**ACTIVE MEMORY**

<table>
<thead>
<tr>
<th>BYTE ADDRESS</th>
<th>0</th>
<th>8K</th>
<th>16K</th>
<th>24K</th>
<th>32K</th>
<th>40K</th>
<th>48K</th>
<th>56K</th>
</tr>
</thead>
<tbody>
<tr>
<td>OVERLAY MAP, FORTRAN STACK, ETC.</td>
<td>5.6</td>
<td>ROOT SEGMENT COMMONS</td>
<td>6.0</td>
<td>ROOT SEGMENT CODE &amp; FORTRAN LIBRARY</td>
<td>11.9</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**"HIGH CORE"**

<table>
<thead>
<tr>
<th>56K</th>
<th>64K</th>
<th>72K</th>
<th>80K</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISPLAY FILE FOR REFRESHED GRAPHICS</td>
<td>SCRATCH AREA</td>
<td>ARCHIVAL DEFINITION</td>
<td>BFILE</td>
</tr>
<tr>
<td>8.0</td>
<td>2.5</td>
<td>4.0</td>
<td></td>
</tr>
</tbody>
</table>

---

**FIGURE 2-10**

**Exemplary Memory Allocation for PDP-11/40**
arrays.

* *

2.5.4 ALTERNATIVE ORGANIZATIONS FOR MINICOMPUTERS

The minimal host system for PADL is, in our judgment,
1) a minicomputer with 64 Kbytes of addressable storage,
2) a disc file whose capacity is at least 256 Kbytes, and
3) a graphic display device which does not require a display file, e.g. a storage tube terminal such as the Tektronix 4010.

This minimal host system has disadvantages. It may not be possible, for example, to compile and link PADL on such a system, but this limitation need not be serious if the host is a satellite to a larger system. The data accessing problem is more serious. If working data files must be stored on disc, the processor is likely to run very slowly. Some kind of virtual memory or data buffer scheme (e.g. a mini version of the High Core discussed above) seems almost essential.

A "maximal" minicomputer host system probably should be based on a multi-tasking operating system capable of servicing a number of 64K storage blocks. This should reduce significantly the overlaying overhead by permitting more task-controlled code to be kept in the computer's main store.

* *

2.5.5 THE DESIGN OF OVERLAY STRUCTURES

"Overlaying" is a virtual-memory technique in which program segments are loaded into core "when needed", according to some (operating system dependent) strategy. Segment loading causes over-writing of a core region and loss of the information previously stored in the region.

Two general design problems are raised by overlaying.

1) The design of "modular" programs that make efficient overlaying possible. Modularity is an elusive notion, related to the way a program is partitioned into subroutines, and to the flow of control, i.e. to the invocation sequences.
2) The design of overlay structures, which is the subject of this section.

The design of overlay structures amounts to grouping routines into segments, and grouping segments into overlays, subject to two basic considerations.

1) The program should execute correctly. As execution proceeds one must always be able to load the required program segments without over-writing information that is needed at a further processing stage.

2) The time penalty associated with overlaying should not be excessive. This implies that "thrashing" must be avoided, i.e. that requests for segment loading must not be too frequent.

Overlay design depends strongly on the facilities for automatic overlaying offered by particular operating systems. We shall not attempt to discuss general principles; rather, we shall focus on the RT-11 environment. This will give the reader an indication of the constraints under which the PADM system was designed, and it will serve also as an "almost worst case" study because the overlaying facilities of RT-11 are rudimentary (but relatively fast at execution time).

The overlaying facilities of RT-11 may be characterized as follows.

1) Core is divided into a number of regions. The length of each region is determined by the linker; it is the length of the longest segment of code associated with the region. Region boundaries are fixed, i.e. they do not vary dynamically.

2) Each segment is associated with a region, and loaded into core whenever one of the segment's routines is invoked (provided that the segment is not already in core).

3) When a routine is executing, its whole "return path" must be in core. Thus, for example, suppose that the main program calls a routine A, which in turn calls B; when B is executing the main program and subroutines A and B must be in core simultaneously.

We shall introduce some terminology by means of the example shown in Fig. 2-11. The "calling structure data" of Fig. 2-11a has the obvious meaning; for example, the main program, DOCAN, calls the three subroutines SENDIT, FILLIT, and MOVCAN, and so forth. Fig. 2-11b shows the corresponding
CALLING STRUCTURE DATA

DOCAN CALLS: SENDIT FILLIT MOVCAN
SENDIT CALLS: MOVCAN DATEIT LABELON
FILLIT CALLS: MOVCAN COVER PEASIN
MOVCAN CALLS:
DATEIT CALLS: ROTATE CANOK
LABELON CALLS: ROTATE CANOK
ROTATE CALLS:
CANOK CALLS:
COVER CALLS: CRIMP CANOK
CRIMP CALLS: CANOK
PEASIN CALLS: CANOK

CALLING TREE

DOCAN .... MOVCAN
.... FILLIT .... PEASIN .... CANOK
..... COVER .... CANOK
.......
..... MOVCAN
.... SENDIT .... LABELON .... CANOK
..... ROTATE
..... DATEIT .... CANOK
..... ROTATE
..... MOVCAN

PATH TABLE

<table>
<thead>
<tr>
<th>NAME</th>
<th>LENGTH</th>
<th>1234567890</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILLIT</td>
<td>600</td>
<td>.XXXXX.....</td>
</tr>
<tr>
<td>DOCAN</td>
<td>350</td>
<td>XXXXXXXXXX</td>
</tr>
<tr>
<td>SENDIT</td>
<td>175</td>
<td>.... X.XXX</td>
</tr>
<tr>
<td>CANOK</td>
<td>105</td>
<td>.XXX. X. X .</td>
</tr>
<tr>
<td>ROTATE</td>
<td>100</td>
<td>.... X. X .</td>
</tr>
<tr>
<td>COVER</td>
<td>30</td>
<td>.XX.......</td>
</tr>
<tr>
<td>CRIMP</td>
<td>20</td>
<td>.... X.....</td>
</tr>
<tr>
<td>PEASIN</td>
<td>20</td>
<td>.X.........</td>
</tr>
<tr>
<td>MOVCAN</td>
<td>15</td>
<td>X...X....X</td>
</tr>
<tr>
<td>DATEIT</td>
<td>15</td>
<td>..... XX.</td>
</tr>
<tr>
<td>LABELON</td>
<td>15</td>
<td>..... XX...</td>
</tr>
</tbody>
</table>

FIGURE 2-11

Data Useful for Planning Overlays
(Partial sample of "O" Program Output)
"calling tree". (Note that several nodes in the "tree" may have the same name.) Each path from the root to a leaf represents a possible invocation sequence, although in practice some of these may never be executed. To each path we can associate a length which is the sum of the lengths of the various routines corresponding to the nodes in the path.

The "return path" restriction of RT-11 (characteristic 3 above) implies that the whole path from a root to a leaf must be in core when the leaf is executing. This implies in turn that the maximum path length determines the minimal amount of core required to run a particular program, and therefore that the program designer should strive for small maximal path lengths.

Core partitioning into regions and assignment of segments to regions are specified by the user of RT-11. Therefore the user has the responsibility for enforcing the "return path" rule, as well as the responsibility for ensuring that no needed information is over-witten when segments are brought into core. In a program of PADL's size -- PADL has approximately 300 routines, and a maximal path length of 13K bytes -- the above responsibilities constitute a heavy burden, especially when the system is being developed and overlay structures must change accordingly.

We used the following approach in the design of PADL's overlay structures.

1) PADL has a large amount of global data such as object definition strings, boundary files, and so on. Because global data will be destroyed if overlaid, we made all these data resident either in low or in high core.

2) We assigned segments to regions in such a way that no two segments in a region may contain routines which lie in the same invocation path.

We found that segment assignments consistent with the above rule were difficult to determine manually, and we wrote a program called "O" which does a good part of the overlay design automatically. The input to "O" is calling structure data similar to that shown in Fig. 2-11a together with routine length data. The program uses simple heuristics to produce overlay structures whose total memory occupancy is close to minimal.

"O" does not address thrashing problems, and thus some human tuning of the structures produced by "O" is often required. Tuning is a relatively delicate operation. For
example, it involves determining those routines which are often called as a group and assigning them to a segment. To assist human overlay designers "0" produces calling trees and path tables such as those shown in Fig. 2-11b and 2-11c.

We shall supply calling trees and path tables for the export version of PADL to facilitate the design of overlays in systems other than RT-11.

* *

2.6 COMMISSIONING THE PROCESSOR

Before the PADL processor can be commissioned on another computer system, system dependent code must be identified and modified. We shall explain PADL's known system dependencies in general terms in 2.6.1 below. (Other documents in the SD and CRDM series deal with specific dependencies.) We shall also suggest in 2.6.2 a commissioning procedure that should help to isolate problems.

* *

2.6.1 SYSTEM DEPENDENCIES

PADL's system dependencies fall into three general categories.

1) Low level machine and device dependencies.

Some routines -- graphic device drivers, HICAAR, and bit manipulators -- are coded in assembly language. Other routines, in particular those that interface to assembly language routines, are often system dependent because they depend on the behavior of lower level system dependent routines.

2) Non-standard FORTRAN library routines and statements.

Often these are operating system dependent, as in the case of disk file handlers and time and date services, but in a few cases facilities are used that are not generally available in FORTRAN.

3) Higher level system dependencies.

The most important of these are the storage and control path constraints that affect overlaying; see relevant subsections in 2.5. There are also a few miscellaneous dependencies, e.g. the assumption of ASCII character
coding and the use of device dependent control parameters in a few graphic output routines.

* *

The Appendix contains a closely indicative list of system dependent routines together with very terse comments on the nature of the dependency. Definitive information is supplied in SD-99.

IMPORTANT NOTE

The processor as ordinarily supplied will include the subroutine and utility packages cited in 2.4 plus routines to service, in a PDP-11/RT-11 environment, a Tektronix 4010 storage tube terminal and a Digital Equipment Corporation VT-11 refreshed graphics terminal. Routines to service a Versatec 2160A printer/plotter can be made available to those who hold appropriate software licenses from Versatec.

* * *

2.6.2 A COMMISSIONING PROCEDURE

The approach we recommend for installing the PADL processor is outlined below as a sequence of nine steps. The philosophy is obvious: separate the processor into manageable subsystems and commission it "bottom-up", testing carefully as you proceed.

1) Acquire and install a FLECS translator; see Section 2.4.1.

2) Acquire (e.g. write) and test the assembly language routines (except for the HICAAR system) and the low level device dependent routines. Many of these may already be available in other systems, but small changes may be needed to conform to the PADL processor's protocols. Test programs for these routines -- especially the graphic output routines -- should be designed with Step 7 below in mind.

3) Acquire and test the extended data storage routines which are to replace the HICAAR system.

4) Compile and test each of the subroutine packages -- CPAK, STGPAK, etc. -- using the test routines supplied with them. The test routines we provide are not exhaustive, and users are encouraged to familiarize themselves with each package by designing more extensive tests.
5) Separate the PADL processor into its four major components -- the input processor, the boundary evaluator, the D&T processors, and the graphic processors -- and design (if necessary) appropriate overlays by using the calling trees and path tables provided with the PADL processor. The major components can be installed in stages (steps 6 - 9 below) if minor temporary changes are made to prevent the System Monitor and Custodian from calling routines or referencing data that are not yet available.

6) Install the input processor. It provides ample diagnostic printout facilities to verify the correctness of object trees and tables.

7) Install the boundary evaluator, its preprocessor (PROCDEF), the definition writer (MAKDEF), and the BFILE system. The boundary evaluator also provides ample diagnostic and trace facilities. Unfortunately, detailed traces are tedious to verify for any but the simplest objects. It is therefore advantageous to provide a crude graphic output facility to produce "pictures" of the data in the b-file.

8) Install and test the graphic output processor.

9) Install and test the D&T processors.

* * *
3. DOCUMENTATION

3.1 A HIERARCHICAL VIEW OF GEOMETRIC MODELLING

Our approach to documentation follows closely our general modelling philosophy. Readers will find reports in the SD series easier to read and easier to place into a unified perspective if they understand from the outset our view of the modelling activity.

We approach geometric modelling in two stages. In the first stage we define a mathematical modelling space by choosing mathematical entities which capture the relevant aspects of the real-world entities under study. The mathematical entities appropriate for studying 3-D industrial geometry are regular subsets of E3 (Euclidean 3-D space). Explicit acknowledgement of such a modelling space allows us to make general statements about elements in the space, and to define general properties in a "representation-free" manner.

In the second stage we introduce a mechanism for designating individual elements of the space (e.g. sets which correspond to particular PADL-like blocks or cylinders) by appropriate symbol structures. Such a mechanism may be viewed as a mapping which associates symbol structures to some of the elements of the mathematical modelling space. We shall term the mapping a "representation scheme", and the structures "representations". The semantics of representations may be described precisely in standard mathematical terms by appealing to concepts pertinent to the mathematical modelling space.

To deal with practical systems one usually needs a hierarchy of representations linked through mappings. High level representations, which are overtly mathematical in style, typically are used to define and analyze a system's major functions, whereas low level representations are "detailed" and define the "implementation" of their high level counterparts. We shall call these two types "mathematical" (or "logical") representations, and "computational" representations. (The latter usually are data structures in some programming language.)

* 

Representation schemes at all levels must be DESIGNED; they are not immutable or natural artifacts. This fact has two important implications.
(1) In general, one should deal only with representation schemes that are "complete" (or unambiguous). This means that a particular representation should designate a unique entity in the mathematical modelling space.

(2) Representation schemes must be designed with specific classes of applications in mind; "universally effective" representations are a chimera. If the completeness property of (1) obtains, however, entities can be RE-REPRESENTED (at least in principle) to match different classes of applications.

The PADL system illustrates both principles nicely. The (complete) constructive "solid" representations which we regard as primary are convenient for dealing with such "volumetric" phenomena as material removal and spatial interference testing. They are not well suited to graphics, however, and thus a (complete) re-representation in terms of bounding surfaces is provided.

* * *

Models and representations deal with entities (data); we can also define transformations on entities. Mathematical functions may be defined at the top level in a representation-free manner and mapped onto a hierarchy of algorithms which operate on representations at various levels.

* * *

3.2 THE APPROACH TO DOCUMENTATION

Documentation of PADL technology falls into the following categories.

1) Explications of basic theory.

2) Discussions of PADL's potential role in design and manufacturing automation.

3) Users' guides to the PADL language and processor.

4) PADL language specifications.

5) PADL processor design and maintenance guides.

SD reports deal only with the last category; categories 1 - 4 are covered elsewhere [1], [2], [4], [15]-[18].
This document, SD-01, serves as a general introduction to the series. The internal organization of most of the higher numbered SD's reflects the modelling hierarchy discussed earlier, in that most SD's contain a section that describes the relevant mathematical representations or algorithms, and a subsequent section that describes computer representations or algorithms.

Computer representations or algorithms are viewed as particular implementations of their mathematical counterparts; the latter are "explained" by appealing to theoretical notions. SD reports do not develop theory. Rather, they use it to define the semantics of representations and algorithms. Since SD reports are oriented toward system programmers as well as designers, we shall usually present mathematical material informally.

Our main reasons for emphasizing mathematical representations and algorithms in the processor's documentation are as follows.

1) We do not claim that our implementations are optimal, but we do claim that they are well defined. The defining media -- mathematical representations and algorithms -- offer an unequivocal starting point for improvements that others may wish to effect.

2) PADL's basic theory and most of the processor's high-level algorithms depend on neither the particular solid primitives and rigid transformations available in the PADL language nor on the manner in which these are implemented as computer representations and algorithms.

3) While the low-level representations and algorithms of the 1.0n processor are tailored to the primitives and rigid transformations available in the PADL-1.0 language, the methodology we used to provide low-level facilities is quite general.

Fact 2) just above virtually guarantees that PADL may be extended to cover any useful class of objects describable through constructive solid geometry on regular sets. It is important to realize, however, that extensions will require design effort both at the (mathematical) representational level and at the implementational level. The SD reports foster extensibility studies by discussing issues and methodologies at a level of generality beyond that necessary to describe the current processor's implementation.

* *
3.3 THE PADL/SD SERIES OF REPORTS

This document, SD-01, serves as a general overview and introduction to the SD series. Table 3-1 shows a desirable documentation plan, i.e. a set of SD's sufficient to define the processor fully. We cannot spare the effort to write all of the listed documents, however, and thus we shall endeavor to provide only the starred entries in Table 3-1. We believe that these, plus the very extensive comments in the processor's source code, will provide adequate documentation for readers who have had some formal training in computer science/engineering (e.g. sufficient to understand the PADL Translator from the source code comments, given that it employs standard precedence-grammar methods [19].)
| Table 3-1: A Desirable Documentation Plan for the PADL-1.0/n Processor |

<table>
<thead>
<tr>
<th>Numbers</th>
<th>Title</th>
<th>Authors</th>
</tr>
</thead>
<tbody>
<tr>
<td>02</td>
<td>Simple Geometric Entities</td>
<td>RBT</td>
</tr>
<tr>
<td>03</td>
<td>Numerical Geometry Routines</td>
<td></td>
</tr>
<tr>
<td>04</td>
<td>Unassigned</td>
<td></td>
</tr>
<tr>
<td>05</td>
<td>System and Application Controllers (Monitors)</td>
<td>EEH &amp; HBV</td>
</tr>
<tr>
<td>06</td>
<td>Constructive Representations</td>
<td>WBF &amp; EEH</td>
</tr>
<tr>
<td>07</td>
<td>The PADL Translator</td>
<td>WBF</td>
</tr>
<tr>
<td>08</td>
<td>Unassigned</td>
<td></td>
</tr>
<tr>
<td>09</td>
<td>The Boundary Evaluator</td>
<td>HBV &amp; RBT</td>
</tr>
<tr>
<td>10</td>
<td>Boundary Representations</td>
<td>GA et al.</td>
</tr>
<tr>
<td>11</td>
<td>The DTA Analyzer</td>
<td>EEH et al.</td>
</tr>
<tr>
<td>12</td>
<td>Graphic Representations</td>
<td>JM</td>
</tr>
<tr>
<td>13</td>
<td>The Graphic Output Processor</td>
<td>JM</td>
</tr>
<tr>
<td>14</td>
<td>The Drawing File</td>
<td>WAH</td>
</tr>
<tr>
<td>15</td>
<td>The Drawing Editor 8 Postprocessors</td>
<td>WAH</td>
</tr>
<tr>
<td>99</td>
<td>PADL-1.0/2.n Status and Commissioning Information</td>
<td>WBF et al.</td>
</tr>
</tbody>
</table>
REFERENCES


APPENDIX

SYSTEM DEPENDENT ROUTINES IN THE PADL-1.0/n PROCESSOR <**>

<table>
<thead>
<tr>
<th>Routine</th>
<th>Subsystem</th>
<th>Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASSIGN</td>
<td>FORTRAN Library</td>
<td>OS dependent disk file routine.</td>
</tr>
<tr>
<td>BFILE</td>
<td>Subroutine pkg.</td>
<td>See SD-10 and the BFILE listings.</td>
</tr>
<tr>
<td>CLOCK</td>
<td>Utility</td>
<td>Uses OS dependent routine.</td>
</tr>
<tr>
<td>CLOSE</td>
<td>FORTRAN Library</td>
<td>OS dependent disk file routine.</td>
</tr>
<tr>
<td>CHOUT</td>
<td>Graphic</td>
<td>Graphic device dependent; assembly language.</td>
</tr>
<tr>
<td>CURSOR</td>
<td>Graphic</td>
<td>Graphic device dependent; assembly language.</td>
</tr>
<tr>
<td>DATE</td>
<td>FORTRAN Library</td>
<td>OS dependent routine.</td>
</tr>
<tr>
<td>GARC</td>
<td>Graphic</td>
<td>Calls graphic device dependent routines.</td>
</tr>
<tr>
<td>GETLIN</td>
<td>Input proc.</td>
<td>Uses special character; see listing.</td>
</tr>
<tr>
<td>GFILES</td>
<td>Graphic</td>
<td>Calls OS dependent routines.</td>
</tr>
<tr>
<td>GLINE</td>
<td>Graphic</td>
<td>Calls graphic device dependent routines.</td>
</tr>
<tr>
<td>GPAK</td>
<td>Subroutine pkg.</td>
<td>See [10] and the GPAK listings.</td>
</tr>
<tr>
<td>GRAFIX</td>
<td>Graphic</td>
<td>Graphic device dependent; assembly language.</td>
</tr>
<tr>
<td>GSPLIN</td>
<td>Graphic</td>
<td>Calls graphic device dependent routines.</td>
</tr>
<tr>
<td>CTEXT</td>
<td>Graphic</td>
<td>Calls graphic device dependent routines.</td>
</tr>
<tr>
<td>HICAAR</td>
<td>Utility pkg.</td>
<td>See [13] and the HICAAR listings; assembly language.</td>
</tr>
<tr>
<td>HICOR</td>
<td>Utility (suite)</td>
<td>Uses HICARR; see listings.</td>
</tr>
<tr>
<td>IAND</td>
<td>Graphic</td>
<td>Assembly language routine.</td>
</tr>
<tr>
<td>IBYTE</td>
<td>Graphic</td>
<td>Assembly language routine.</td>
</tr>
<tr>
<td>INITVT</td>
<td>Graphic</td>
<td>Graphic device dependent routine.</td>
</tr>
<tr>
<td>IOSERV</td>
<td>Input proc.</td>
<td>Calls OS dependent routines.</td>
</tr>
<tr>
<td>ISHIFT</td>
<td>Graphic</td>
<td>Assembly language routine.</td>
</tr>
<tr>
<td>JTYPE</td>
<td>Input proc.</td>
<td>Assumes ASCII character coding.</td>
</tr>
<tr>
<td>LAYOUT</td>
<td>Graphic</td>
<td>Uses device dependent parameters.</td>
</tr>
<tr>
<td>LINKVT</td>
<td>Graphic</td>
<td>Graphic device dependent; assembly language.</td>
</tr>
<tr>
<td>MIXBLD</td>
<td>Graphic</td>
<td>Uses device dependent parameters.</td>
</tr>
</tbody>
</table>

<**> The list shown here may not be absolutely complete or up-to-date; see SD-99 for definitive information. See also the "Important Note" in Section 2.6.1.
<table>
<thead>
<tr>
<th>Routine&lt;**&gt;</th>
<th>Subsystem</th>
<th>Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRSDAS</td>
<td>Input proc.</td>
<td>Uses OS dependent routines.</td>
</tr>
<tr>
<td>RESTVT</td>
<td>Graphic</td>
<td>Graphic device dependent; assembly language.</td>
</tr>
<tr>
<td>SETTRC</td>
<td>Input proc.</td>
<td>Uses OS dependent routines.</td>
</tr>
<tr>
<td>STCPAK</td>
<td>Subroutine pkg.</td>
<td>See [9] and the STCPAK listings.</td>
</tr>
<tr>
<td>TEKRES</td>
<td>Graphic</td>
<td>Graphic device dependent; assembly language.</td>
</tr>
<tr>
<td>TIME</td>
<td>FORTRAN Library</td>
<td>OS dependent routine.</td>
</tr>
<tr>
<td>TKARC</td>
<td>Graphic</td>
<td>Graphic device dependent routine.</td>
</tr>
<tr>
<td>TKDRAW</td>
<td>Graphic</td>
<td>Graphic device dependent routine.</td>
</tr>
<tr>
<td>TKLNE</td>
<td>Graphic</td>
<td>Graphic device dependent routine.</td>
</tr>
<tr>
<td>TKSPLN</td>
<td>Graphic</td>
<td>Graphic device dependent routine.</td>
</tr>
<tr>
<td>TKTEXT</td>
<td>Graphic</td>
<td>Graphic device dependent routine.</td>
</tr>
<tr>
<td>VTARC</td>
<td>Graphic</td>
<td>Graphic device dependent routine.</td>
</tr>
<tr>
<td>VTDRAW</td>
<td>Graphic</td>
<td>Graphic device dependent routine.</td>
</tr>
<tr>
<td>VTLINE</td>
<td>Graphic</td>
<td>Graphic device dependent routine.</td>
</tr>
<tr>
<td>VTSPLN</td>
<td>Graphic</td>
<td>Graphic device dependent routine.</td>
</tr>
<tr>
<td>VTTEXT</td>
<td>Graphic</td>
<td>Graphic device dependent routine.</td>
</tr>
</tbody>
</table>

<**> Prefixes TK... and VT... denote respectively Tektronix and Digital Equipment Corporation CRT displays.